

1 Introduction

1.1 Trois points de vue sur la notation objet

- Abstraction (type abstrait) - point de vue structure
 - Cache les structures (ex : pile) par les manipulations légales (ex : empiler, dépiler)
 - On parle *d'encapsulation* de l'objet
- Réseaux sémantiques (Intelligence artificielle) - point de vue concept
 - Sous classe SC *est une sorte de classe* C
 - Objet O *est un (est une instance) de* C
- Langage d'acteurs (Intelligence artificielle) - point de vue dynamique
 - l'objet O reçoit un message et y réagit
 - Système multi-agents

1.2 Trois grandes classes d'applications

- La conception = représenter le monde réel pour l'informatiser
 - Représente les entités (classe) et les liens (héritage) naturellement
 - Visualisation facile (graphe des classes), synthèse et modifications
- La programmation = le codage
 - La programmation jaillit de la modélisation
 - Les fonctions sont relatives aux données
 - Un même nom de fonction pour même opération, même si codage différent (langage abstrait)
- Les bases de données = la persistance
 - Pourquoi utiliser des tables ?
 - Objets persistants

1.3 Principes

- Principe 1 : on raccroche à la structure de données (champs), les codes qui peuvent les manipuler (méthode) -> classe
- Principe 2 : on hiérarchise les classes -> héritage
- Principe 3 : on assure la sécurité des accès aux données et méthodes sous forme d'autorisation ; on gère les erreurs

2 Propriétés des accès

Qui peut accéder aux champs, méthodes et classes ?

2.1 Le constructeur

- Appliqué à une classe retourne un objet de la classe
 - Créer un objet à partir d'une classe
 - Initialiser l'objet par les valeurs adéquates (initialiser après le new les champs)
- Programmer des constructeurs paramétrés
 - (LOLO) : `nomClasse <= new(val1,...,valn)`
 - (en JAVA) : définir des méthodes du nom de la classe
 - * Quand aucun constructeur n'est défini -> constructeur par défaut
 - * Le constructeur par défaut est désactivé dès qu'il existe un constructeur programmé
 - * On peut reconstruire à la main un constructeur sans paramètre
- L'utilisation de *super*
 - `super()` appel le constructeur de la sur-classe
 - `super.méthode(...)` appel de méthode de la classe supérieur

2.2 Accès privé/public

L'accès à un champs peut être :

- **privé** : seules les méthodes de la classe C peuvent utiliser le champs
- **public** : tout code (ayant accès à la classe C) peut l'utiliser
- **protégé** : seules les méthodes de la classe C et de ses sous classes peuvent utiliser le champs

Règles :

- un champs est généralement privé
- -> accès par des méthodes adéquates
- un champs gérant des données d'implémentation est privé

L'accès aux méthodes m d'une classe C peut être :

- **privé** : seules les méthodes de la classe C peuvent utiliser m
- **public** : tout code (ayant accès à la classe C) peut utiliser m
- **protégé** : seules les méthodes de la classe C et de ses sous classes peuvent utiliser m

Remarques :

- *private*, *public*, *protected* (en JAVA) sont appelés des modificateurs (d'accès)
- Accesseurs (conventions)
 - *getVar* : accès lecture à un champs *Var*
 - *setVar* : accès écriture à un champs *Var*

On parle de portée d'une déclaration

2.3 Les paquetages

Un paquetage, en JAVA, n'existe pas en Smalltalk ou C++

- Idées
 - pour de grandes applications
 - regroupe des classes dans un paquetage (*package* en JAVA)
 - Nom de la classe, référé à un paquetage (permet 2 classes de même nom)
 - Les paquetages organisent logiquement les classes dans une hiérarchie ; généralement, l'organisation physique est la même
- Principes du package (logique) et de son positionnement système (physique)
 - Un package est associé à un répertoire : le nom du package est celui du répertoire
 - Une hiérarchie de packages correspond à une hiérarchie de répertoires
 - Dans un package, il y a des classes, et des sous packages (= sous répertoires)
 - Le premier niveau de hiérarchie correspond au nom de l'application ou du propriétaire (ex : loiseau, java)
- Principes de nommage, d'appartenance et de recherche
 - un mécanisme de nommage : *nomPackage.nomSousPackage.NomClasse*

- appartenance :
 - * Par défaut : une classe dans un répertoire appartient au package correspondant
 - * Par déclaration : `package nomPackage.nomSousPackage (1ere instr)`
- recherche : variable `CLASSPATH` (Windows)
- Accès aux classes d'un package
 - les classes (*object*, *String*...) du package *java.lang* sont utilisables dans toute classe
 - pour importer une classe C ou un package P dans un programme : *import C* et *import P.** en tête de fichier
 - Si pas de modificateur de champs et méthode : toutes les classes du package y ont accès
 - Attention : Modificateur d'accès d'une classe C, *public* : la classe est accessible en dehors du package où elle est définie pas de modificateur : seules les classes du même paquetage ont accès à C

3 Les classes pour elles-mêmes

3.1 Retour à l'héritage

- l'autoréférencement
 - pour appliquer une méthode à l'objet
 - pour lever l'ambiguïté entre nom de champs et variable
- L'option *final* (sécurité)
 - une méthode *final* ne peut pas être redéfinie dans une sous classe
 - une classe définie *final* ne peut pas être héritée
 - une variable *final* est une constante

3.2 Classe abstraite

Définition 1 Une classe abstraite est une classe pour laquelle il ne peut exister d'instances.

- elle factorise les champs et méthodes de sous classes
- elle est déclarée avec le mot clef *abstraite* (*abstract*)

Définition 2 Une méthode abstraite d'une classe C est une méthode ne possédant pas de corps. La méthode doit être implémentée dans chaque sous-classe de la classe C

- elle est redéfinie au niveau des sous classes
- vérification à la compilation des redéfinitions obligatoires dans les sous-classes
- elle est déclarée avec le mot clef *abstraite* (*abstract*)
- Vérification : Si C a une méthode abstraite $\Rightarrow C$ est abstraite

3.3 Variables de classe et méthodes de classe

Définition 3 Une variable de classe est une variable rattachée à la classe.

- elle existe donc qu'en un seul exemplaire en mémoire
- elle est déclarée avec le modificateur *static*
- elle peut être privé, public, protégé
- elle est automatiquement créée en mémoire dès que la classe est chargée

Définition 4 Une méthode de classe est une méthode qui s'applique à la classe.

- elle est déclarée avec le modificateur *static*
- elle peut être privé, public, protégé
- elle peut avoir des arguments

3.4 Redéfinition, transtypage, polymorphisme, surcharge

Définition 5 Une méthode est redéfinie si elle est définie dans une classe et (re)définie dans une des classes dont elle hérite

Définition 6 Le transtypage est l'opération qui consiste à changer le type d'un objet. Il existe deux types de transtypage : l'implicite et l'explicite (*cast*). Un objet de classe SC , sous classe de C , est implicitement transtypé en objet de classe C si besoin est. La réciproque est fausse et provoque une erreur.

Définition 7 Le polymorphisme est la capacité à faire correspondre à un même message plusieurs méthodes selon le type de l'objet receveur ; quand l'objet est stocké dans une variable de type général.

Définition 8 *La signature d'une méthode est composé du nom de la méthode, des types des paramètres formels de la méthode. Il y a surcharge (=surdéfinition) lorsqu'une méthode est définie avec plusieurs signatures dans la même classe*

Remarque : souvent en programmation classique, on ne peut pas donner deux noms identiques à des fonctions ou procédures, même si les signatures diffèrent.

Les classes internes :

- Principe : une classe contenue dans une autre classe

Définition 9 *Une classe interne CI (par opposition à classe globale) est une classe déclarée dans une autre classe C*

- Règles
 - Seul C a accès à CI.
 - La création d'une instance de CI ne peut se faire que dans une méthode de C
 - L'instance de CI peut avoir accès aux champs de l'objet O (de C) qui l'a créée
 - Une classe interne CI peut accéder aux champs et méthodes de sa classe
- Intérêt : une classe interne CI est invisible aux autres classes que C

3.5 Héritage multiple et interface

Héritage multiple en C++, mais pas en Smalltalk, ni en JAVA (héritage simple)

Définition 10 *Quand une classe peut être sous classe de plusieurs classes, on dit que l'on a un héritage multiple.*

Problèmes : Soit SC une sous-classe de C1 et C2,

- si deux champs identiques dans super classes, duquel hérite l'objet ?
- si deux méthodes identiques dans super classes, de laquelle hérite l'objet ?

Le concept d'interface (JAVA)

- Un concept
 - uniquement de modélisation de l'héritage multiple

- Une interface est un concept qui définit des comportements sous forme de méthodes abstraites
- Ces comportements doivent être redéfinis dans les classes qui implémentent l'interface
- Caractéristiques d'une interface
 - Comme une classe abstraite, n'a pas d'instance
 - Toutes ses méthodes sont abstraites
 - Ne possède aucun champs
 - Peut avoir des variables dites d'interface (= variables de classes)
 - Une classe peut implémenter (« hériter »), autant d'interfaces que souhaitée
 - Une interface ne peut « hériter » que d'interfaces
- Syntaxe
 - *interface nom_inter { ... }*
 - *class C implements nom_inter { ... }*

Le concept d'interface

- Un concept
 - uniquement de modélisation de la partie "sorte-de" multiple
 - Une interface définit des comportements sous forme de méthodes abstraites
 - Ces comportements doivent être redéfinis dans les classes qui implémentent l'interface
- Caractéristiques d'une interface
 - Comme une classe abstraite, n'a pas d'instance
 - Ne possède aucun champs
 - Une classe peut implémenter, autant d'interfaces que souhaitée
 - Une interface ne peut être-une-sort-de que d'interfaces
- Syntaxe
 - *interface nomInterface { ... }*
 - *classe C implements nomInterface { ... }*

Règles : à la compilation, on vérifie que les classes implémenteuses ont bien les méthodes de l'interface

Note : une interface peut avoir des variables dites d'interface (= variables de classes)

3.6 Gestion des exceptions

- Le problème
 - gestion des erreurs systèmes (erreur de type, erreur d'héritage...)
 - gestion des erreurs prévues
 - l'informatique : la science des erreurs ?
 - temps réel : récupération dans tous les cas (selon T, ...)
- Le principe
 - zone dangereuse : une partie de code où une exception peut avoir lieu
 - la levée d'exception : endroit de la zone dangereuse où est déclenchée l'exception qui fait sortir violemment de la zone dangereuse
 - traitement de l'exception : instructions à exécuter lors d'une levée d'exception, avant de poursuivre le programme

En POO :

- Principes de la zone dangereuse, de la levée d'exception et du traitement de l'exception
 - la levée de l'exception se fait par la création d'un objet qui est issu d'une classe d'exception (Throwable), et qu'on envoie
 - Le traitement de l'exception dépend de la classe dont est issu l'objet qui l'a levée
- Zone dangereuse
 - On pose une zone dangereuse avec le mot clef *try*, l'instruction (ou le bloc) qui suit peut contenir des levées d'exceptions, le traitement des exceptions suit cette instruction.
 - *Try {opération_dangereuse1, opération_dangereuse2;}*
- Le traitement des exceptions se programme de la manière suivante : on précise pour chaque classe d'exception l'instruction (ou le bloc) à effectuer si une exception de cette classe a été levée ; pour cela on fait suivre le mot clef *catch* du nom de la classe de l'exception et d'un nom de variable qui va contenir l'objet exception qui a été créé à l'appel. L'instruction à effectuer suit.
- Lancer (ou lever) une exception (Création)
 - ```
Catch(mon_exception_a_moi2 o2) {Traitement2 ;}
...
// o2 est l objet créé dans une opération dangereuse, il
est de la classe mon_exception_a_moi2, et peut donc être
référéncé sous le nom o2 dans le traitement Traitement2
```



- On peut éventuellement ajouter à la fin du bloc d'exception, une instruction (ou un bloc) à exécuter dans tous les cas. Celle-ci est introduite par le mot clef *Finally*.
- *Finally {est-toujours-exécuté-si-présent;}*
- Il faut évidemment avoir programmer les classes d'exception.
- Pour lever une exception (dans une opération dangereuse), on utilise le mot clef *throw* suivi de l'objet qui est l'exception. *throw new mon\_exception\_a\_moi()*
- la levée d'exception est possible sur tout objet de type sous-classe de *Throwable*, dont deux classes fournies par JAVA sont *Exception* et *Error*
- Si le bloc de traitement d'exception n'est pas dans la même méthode que la levée d'exception, il faut associer à chaque déclaration de méthode que peut traverser l'exception, le nom de la classe d'exception avec le mot clef *throws* suivi du nom des classes d'exceptions qu'elle peut transmettre.
- En SmallTalk, après le traitement d'une exception, on peut si on le souhaite demander à reprendre l'instruction qui a levé l'exception, ou celle qui suit la demande de levée. Ex : *Play, Replay*

